

Lecture #08

Carnegie Mellon University

ADVANCED DATABASE SYSTEMS

OLTP Indexes
(Trie Data Structures)

@Andy_Pavlo // 15-721 // Spring 2019



TODAY'S AGENDA

Index Implementation Issues

Judy Array

ART

Masstree



INDEX IMPLEMENTATION ISSUES

Garbage Collection

Memory Pools

Non-Unique Keys

Variable-length Keys

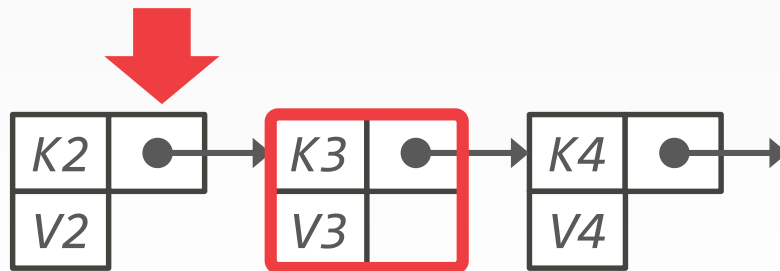
Compression



GARBAGE COLLECTION

We need to know when it is safe to reclaim memory for deleted nodes in a latch-free index.

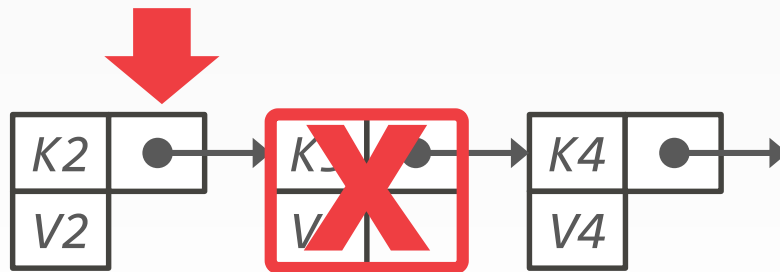
- Reference Counting
- Epoch-based Reclamation
- Hazard Pointers
- Many others...



GARBAGE COLLECTION

We need to know when it is safe to reclaim memory for deleted nodes in a latch-free index.

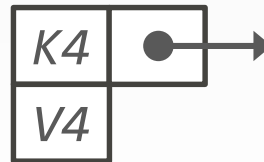
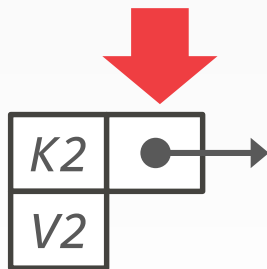
- Reference Counting
- Epoch-based Reclamation
- Hazard Pointers
- Many others...



GARBAGE COLLECTION

We need to know when it is safe to reclaim memory for deleted nodes in a latch-free index.

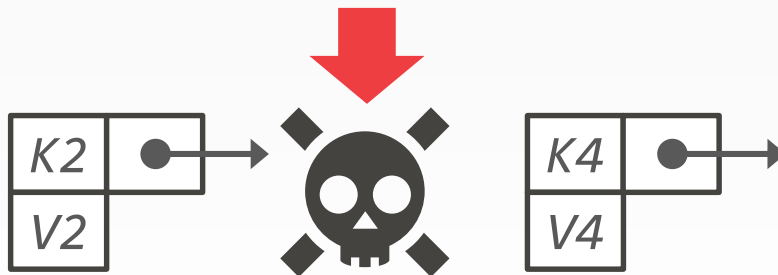
- Reference Counting
- Epoch-based Reclamation
- Hazard Pointers
- Many others...



GARBAGE COLLECTION

We need to know when it is safe to reclaim memory for deleted nodes in a latch-free index.

- Reference Counting
- Epoch-based Reclamation
- Hazard Pointers
- Many others...



REFERENCE COUNTING

Maintain a counter for each node to keep track of the number of threads that are accessing it.

- Increment the counter before accessing.
- Decrement it when finished.
- A node is only safe to delete when the count is zero.

This has bad performance for multi-core CPUs

- Incrementing/decrementing counters causes a lot of cache coherence traffic.

OBSERVATION

We don't actually care about the actual value of the reference counter. We only need to know when it reaches zero.

We don't have to perform garbage collection immediately when the counter reaches zero.

Source: [Stephen Tu](#)

EPOCH GARBAGE COLLECTION

Maintain a global epoch counter that is periodically updated (e.g., every 10 ms).

→ Keep track of what threads enter the index during an epoch and when they leave.

Mark the current epoch of a node when it is marked for deletion.

→ The node can be reclaimed once all threads have left that epoch (and all preceding epochs).

Also known as ***Read-Copy-Update*** (RCU) in Linux.

MEMORY POOLS

We don't want to be calling **malloc** and **free** anytime we need to add or delete a node.

If all the nodes are the same size, then the index can maintain a pool of available nodes.

→ **Insert:** Grab a free node, otherwise create a new one.

→ **Delete:** Add the node back to the free pool.

Need some policy to decide when to retract the pool size.

NON-UNIQUE INDEXES

Approach #1: Duplicate Keys

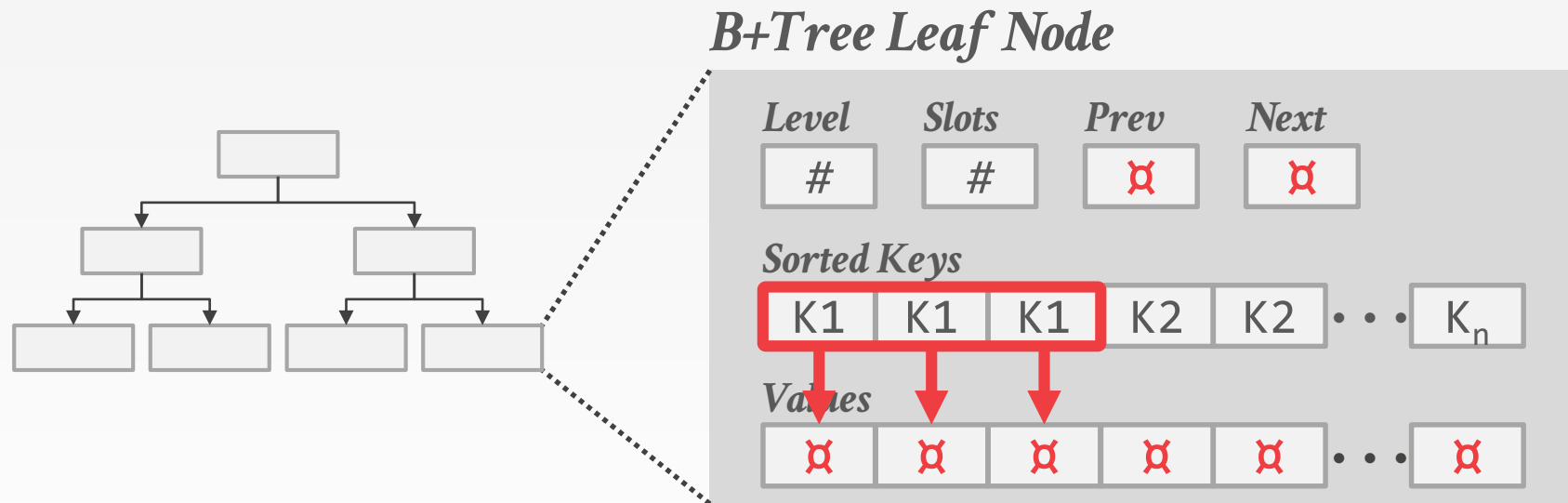
- Use the same node layout but store duplicate keys multiple times.

Approach #2: Value Lists

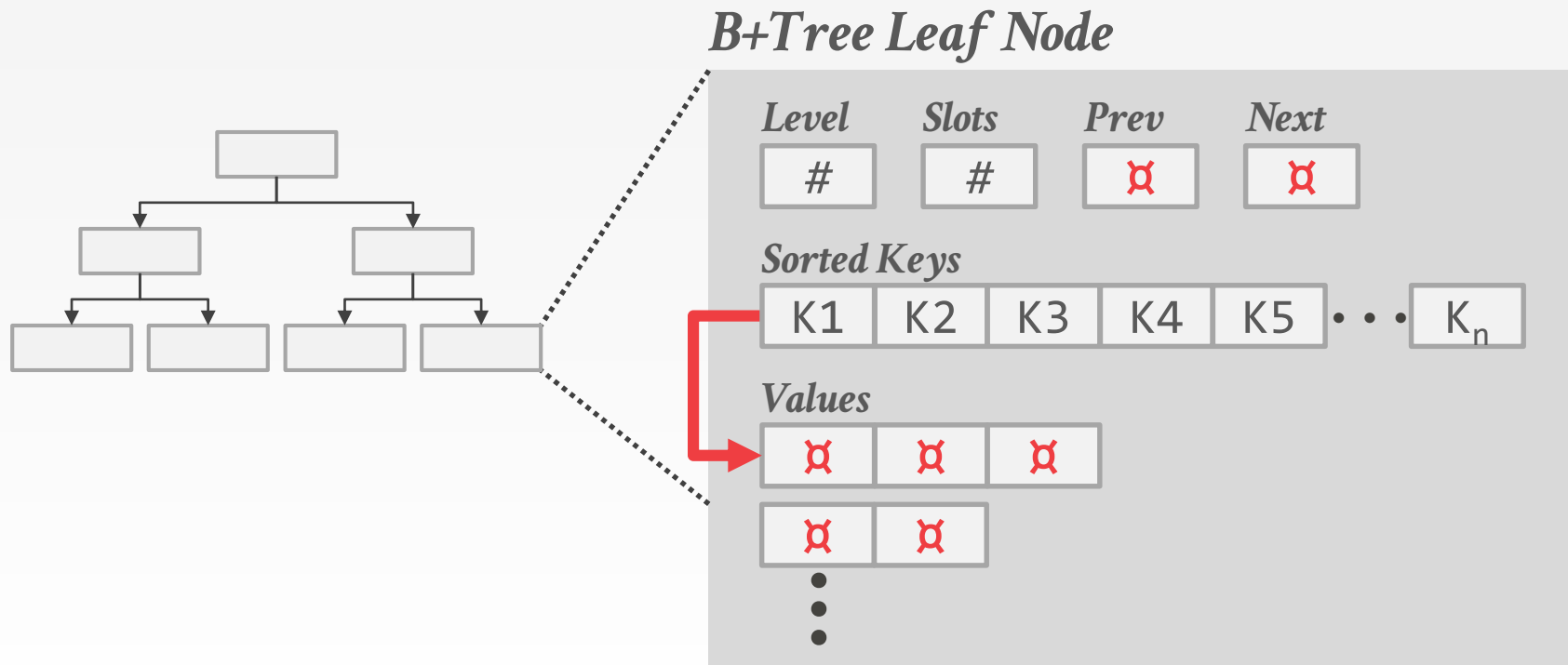
- Store each key only once and maintain a linked list of unique values.



NON-UNIQUE: DUPLICATE KEYS



NON-UNIQUE: VALUE LISTS



VARIABLE LENGTH KEYS

Approach #1: Pointers

→ Store the keys as pointers to the tuple's attribute.

Approach #2: Variable Length Nodes

→ The size of each node in the index can vary.

→ Requires careful memory management.

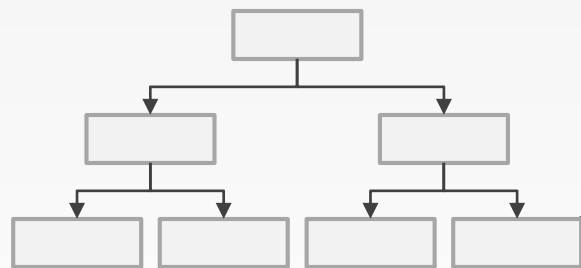
Approach #3: Padding

→ Always pad the key to be max length of the key type.

Approach #4: Key Map / Indirection

→ Embed an array of pointers that map to the key + value list within the node.

KEY MAP / INDIRECTION



B+Tree Leaf Node

<i>Level</i>	<i>Slots</i>	<i>Prev</i>	<i>Next</i>
#	#	✗	✗

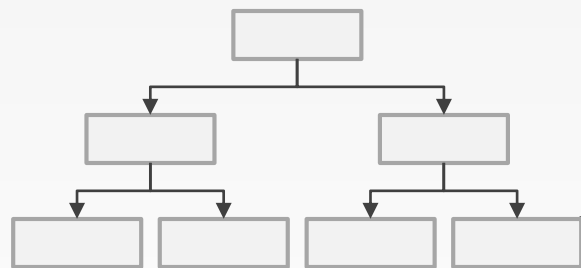
Sorted Key Map

✗	✗	✗	✗
---	---	---	---

Key+Values

		←		Prashanth	V4
Lin	V2	Andy	V1	Obama	V3

KEY MAP / INDIRECTION



B+Tree Leaf Node

<i>Level</i>	<i>Slots</i>	<i>Prev</i>	<i>Next</i>
#	#	⌘	⌘

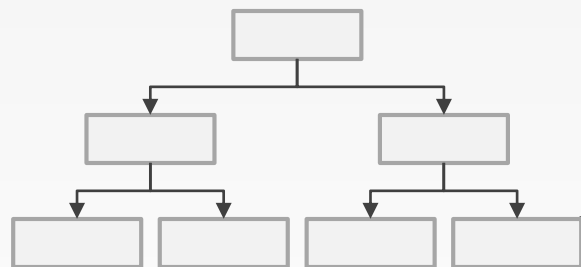
Sorted Key Map

⌘	⌘	⌘	⌘
---	---	---	---

Key+Values

Lin	V2	Andy	V1	Obama	V3
				Prashanth	V4

KEY MAP / INDIRECTION



B+Tree Leaf Node

<i>Level</i>	<i>Slots</i>	<i>Prev</i>	<i>Next</i>
#	#	⌘	⌘

Sorted Key Map

A · ⌘	L · ⌘	O · ⌘	P · ⌘
-------	-------	-------	-------

Key+Values

				Prashanth	V4
Lin	V2	Andy	V1	Obama	V3

PREFIX COMPRESSION

Sorted keys in the same leaf node are likely to have the same prefix.

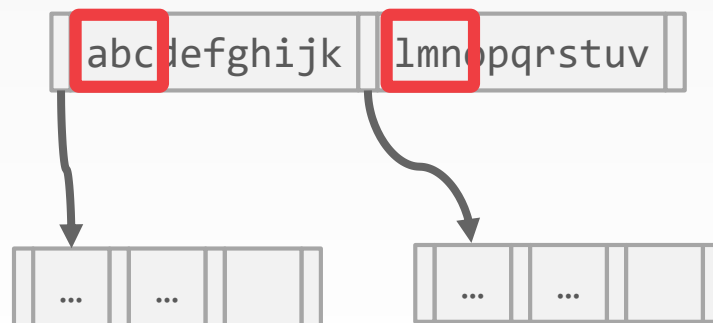
Instead of storing the entire key each time, extract common prefix and store only unique suffix for each key.



SUFFIX TRUNCATION

The keys in the inner nodes are only used to "direct traffic". We don't need the entire key.

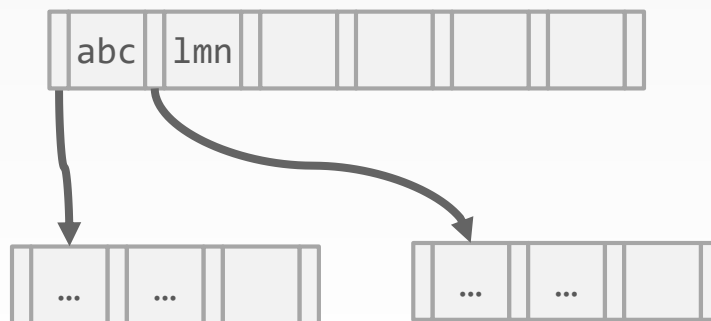
Store a minimum prefix that is needed to correctly route probes into the index.



SUFFIX TRUNCATION

The keys in the inner nodes are only used to "direct traffic". We don't need the entire key.

Store a minimum prefix that is needed to correctly route probes into the index.



OBSERVATION

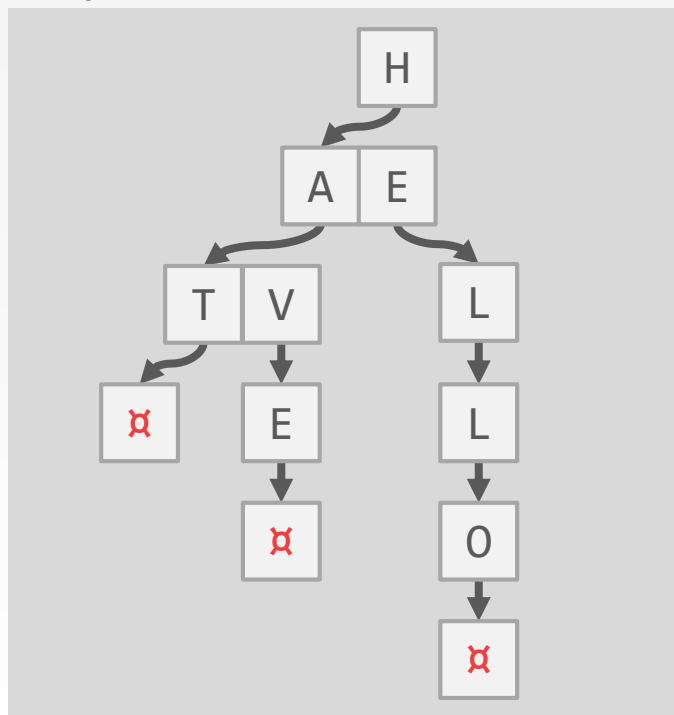
The inner node keys in a B+tree cannot tell you whether a key exists in the index. You always have to traverse to the leaf node.

This means that you could have (at least) one cache miss per level in the tree.



TRIE INDEX

Keys: HELLO, HAT, HAVE

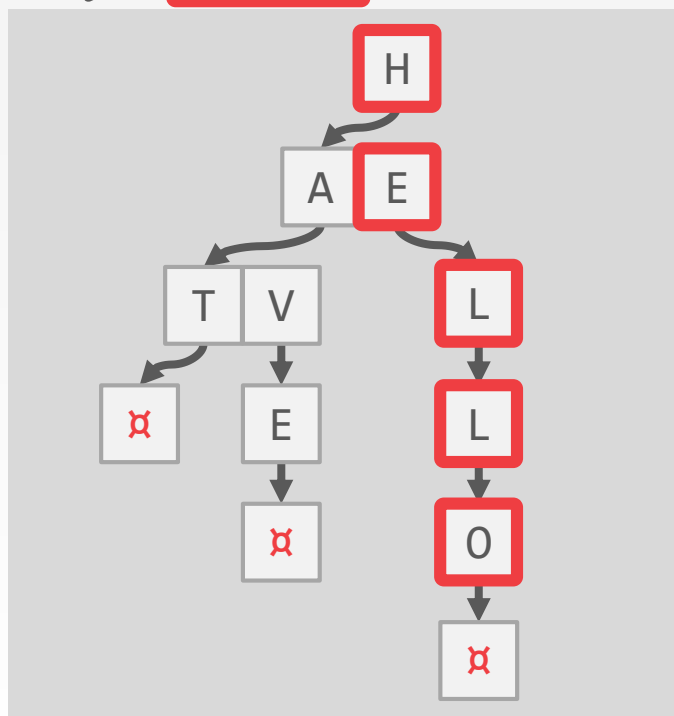


Use a digital representation of keys to examine prefixes one-by-one instead of comparing entire key.

→ Also known as *Digital Search Tree*, *Prefix Tree*.

TRIE INDEX

Keys: **HELLO** HAT, HAVE



Use a digital representation of keys to examine prefixes one-by-one instead of comparing entire key.

→ Also known as *Digital Search Tree*, *Prefix Tree*.

TRIE INDEX PROPERTIES

Shape only depends on key space and lengths.

- Does not depend on existing keys or insertion order.
- Does not require rebalancing operations.

All operations have $O(k)$ complexity where k is the length of the key.

- The path to a leaf node represents the key of the leaf
- Keys are stored implicitly and can be reconstructed from paths.

TRIE KEY SPAN

The span of a trie level is the number of bits that each partial key / digit represents.

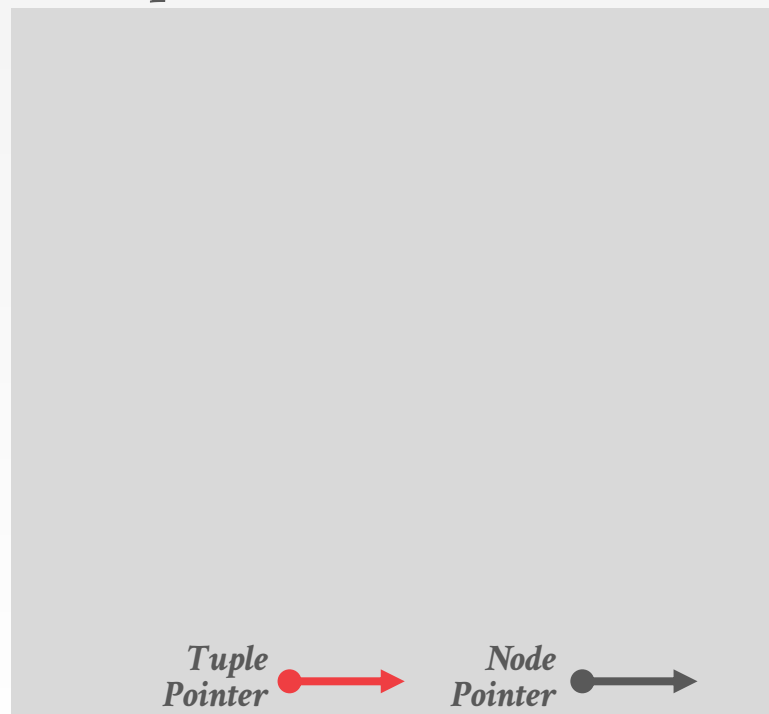
→ If the digit exists in the corpus, then store a pointer to the next level in the trie branch. Otherwise, store null.

This determines the fan-out of each node and the physical height of the tree.

→ n -way Trie = Fan-Out of n

TRIE KEY SPAN

1-bit Span Trie



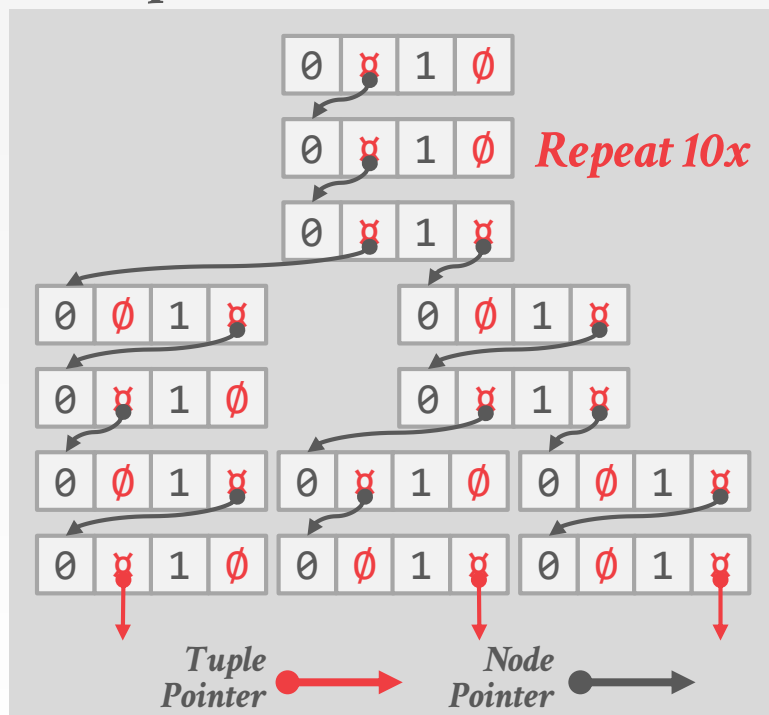
Keys: K10, K25, K31

K10 → 00000000 00001010

K25 → 00000000 00011001

K31 → 00000000 00011111

1-bit Span Trie



Keys: K10, K25, K31

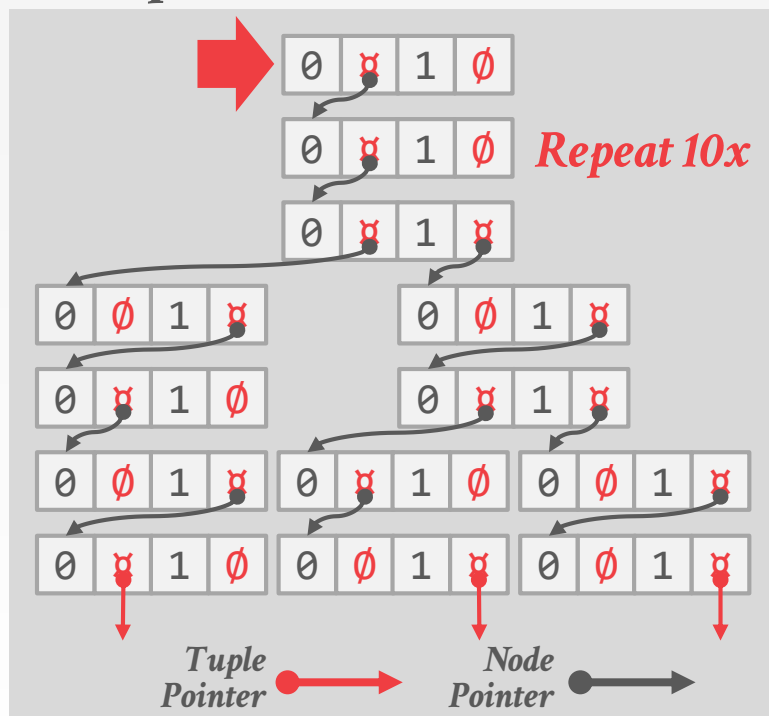
K10→ 00000000 00001010

K25→ 00000000 00011001

K31→ 00000000 00011111

TRIE KEY SPAN

1-bit Span Trie

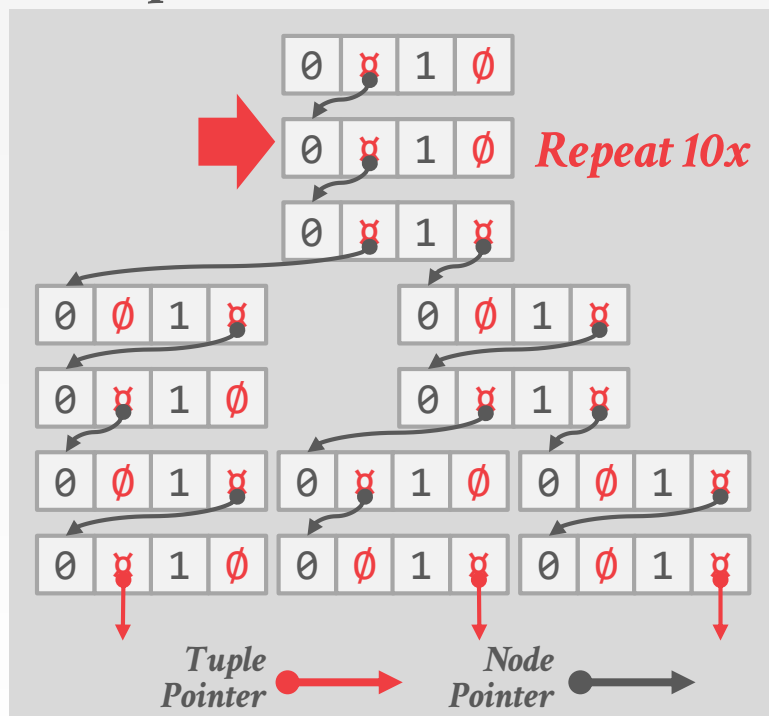


Keys: K10, K25, K31

K10→ 00000000 00001010
 K25→ 00000000 00011001
 K31→ 00000000 00011111

TRIE KEY SPAN

1-bit Span Trie

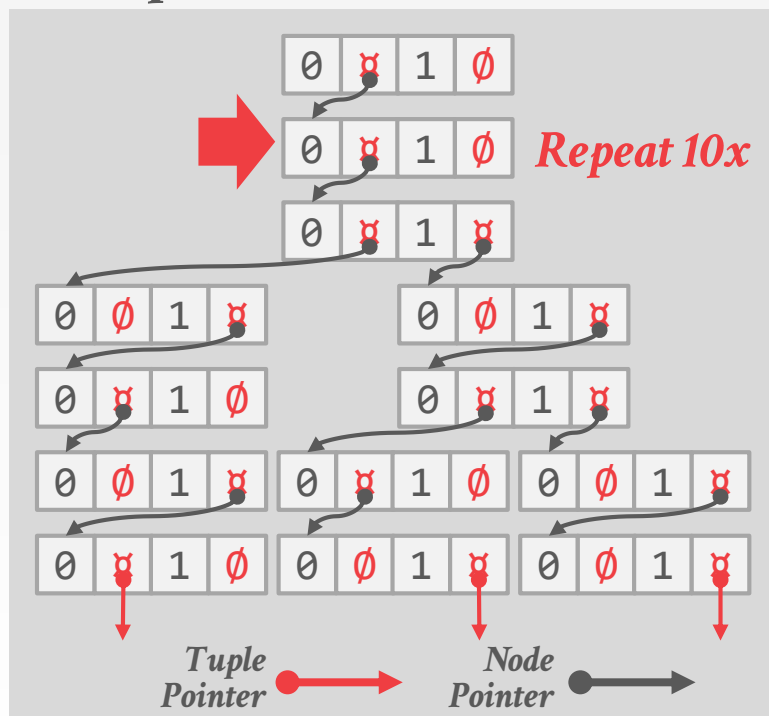


Keys: K10, K25, K31

K10 → 00000000 00001010
 K25 → 00000000 00011001
 K31 → 00000000 00011111

TRIE KEY SPAN

1-bit Span Trie



Keys: K10, K25, K31

K10 → 00000000 00001010
 K25 → 00000000 00011001
 K31 → 00000000 00011111

1-bit Span Trie



K10→	00000000	00001010
K25→	00000000	00011001
K31→	00000000	00011111

1-bit Span Trie



K10 → 00000000 00001010

K25→ 00000000 00011001

K31→ 00000000 00011111

1-bit Span Trie

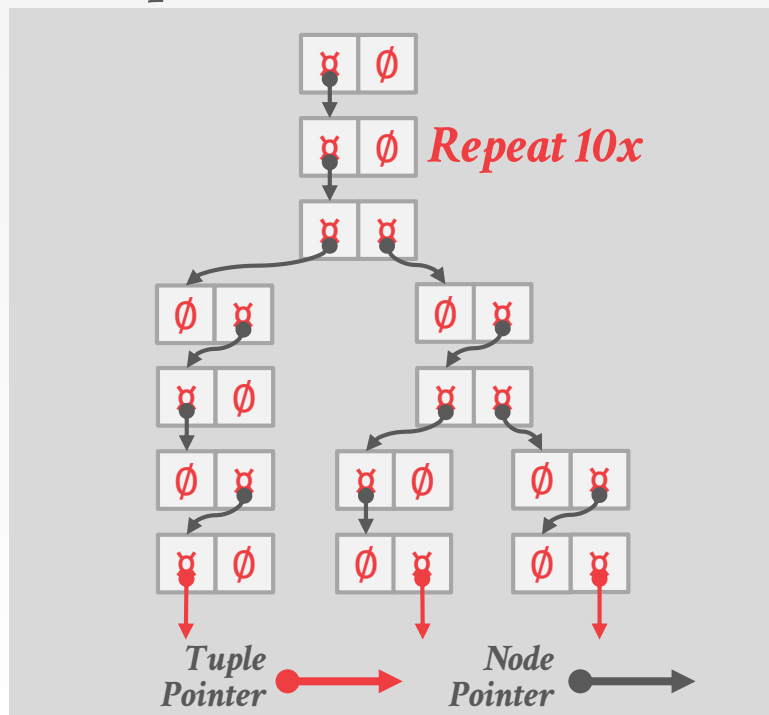


K10→ 00000000 00001010

K25 → 00000000 00011001

K31 → 00000000 00011111

1-bit Span Trie



Keys: K10, K25, K31

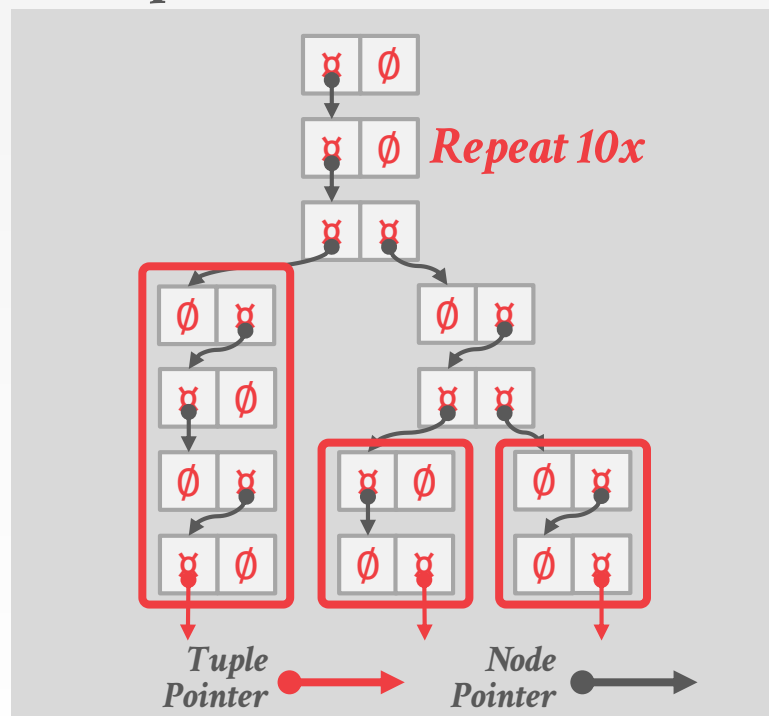
K10→ 00000000 00001010

K25→ 00000000 00011001

K31→ 00000000 00011111

TRIE KEY SPAN

1-bit Span Trie



Keys: K10, K25, K31

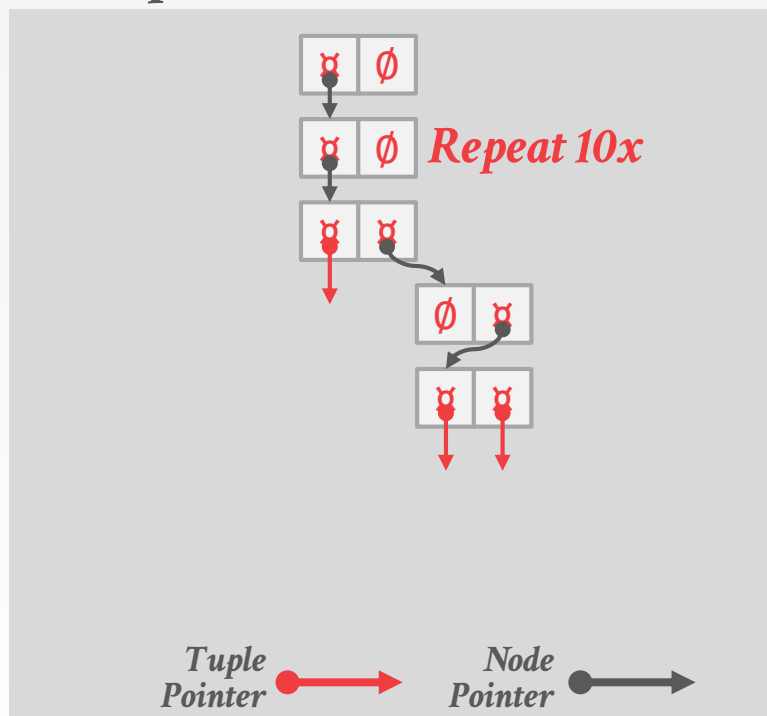
K10 → 00000000 00001010

K25 → 00000000 00011001

K31 → 00000000 00011111

RADIX TREE

1-bit Span Radix Tree



Omit all nodes with only a single child.

→ Also known as *Patricia Tree*.

TRIE VARIANTS

Judy Arrays (HP)

ART Index (HyPer)

Masstree (Silo)



JUDY ARRAYS

Variant of a 256-way radix tree. First known radix tree that supports adaptive node representation.

Three array types

- **Judy1**: Bit array that maps integer keys to true/false.
- **JudyL**: Map integer keys to integer values.
- **JudySL**: Map variable-length keys to integer values.

Open-Source Implementation (LGPL).

Patented by HP in 2000. Expires in 2022.

- Not an issue according to authors.
- <http://judy.sourceforge.net/>

JUDY ARRAYS

Do not store meta-data about node in its header.

→ This could lead to additional cache misses.

Pack meta-data about a node in 128-bit "Judy Pointers" stored in its parent node.

→ Node Type

→ Population Count

→ Child Key Prefix / Value (if only one child below)

→ 64-bit Child Pointer



JUDY ARRAYS: NODE TYPES

Every node can store up to 256 digits.

Not all nodes will be 100% full though.

Adapt node's organization based on its keys.

- **Linear Node:** Sparse Populations
- **Bitmap Node:** Typical Populations
- **Uncompressed Node:** Dense Population



JUDY ARRAYS: LINEAR NODES

Linear Node



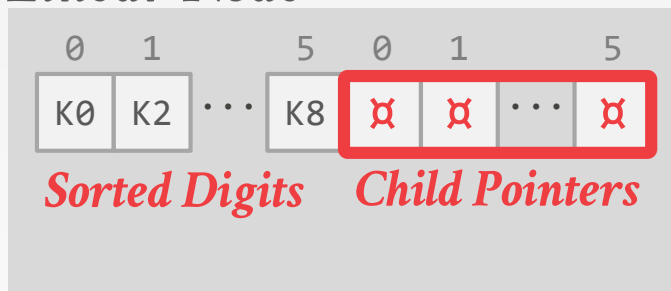
Store sorted list of partial prefixes up to two cache lines.

→ Original spec was one cache line

Store separate array of pointers to children ordered according to prefix sorted.

JUDY ARRAYS: LINEAR NODES

Linear Node



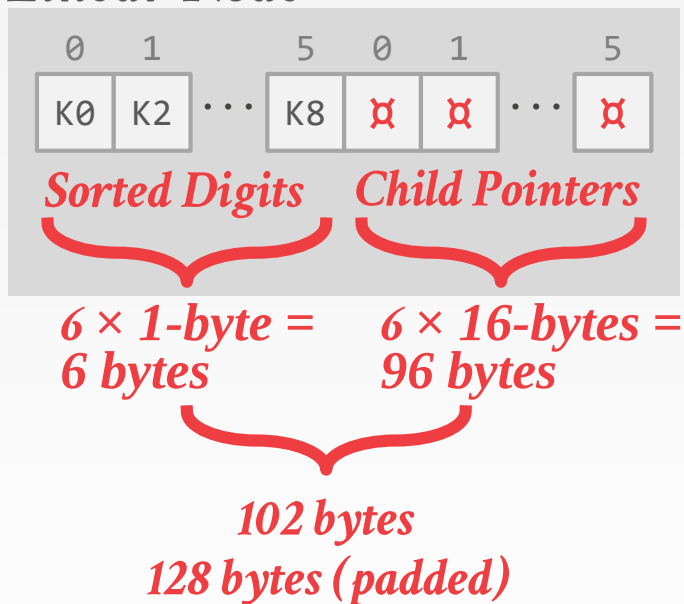
Store sorted list of partial prefixes up to two cache lines.

→ Original spec was one cache line

Store separate array of pointers to children ordered according to prefix sorted.

JUDY ARRAYS: LINEAR NODES

Linear Node



Store sorted list of partial prefixes up to two cache lines.

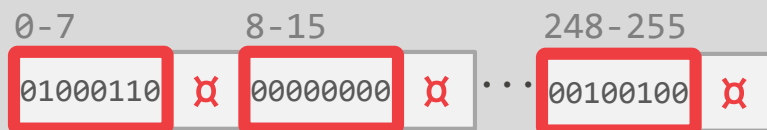
→ Original spec was one cache line

Store separate array of pointers to children ordered according to prefix sorted.

JUDY ARRAYS: BITMAP NODES

Bitmap Node

Prefix Bitmaps



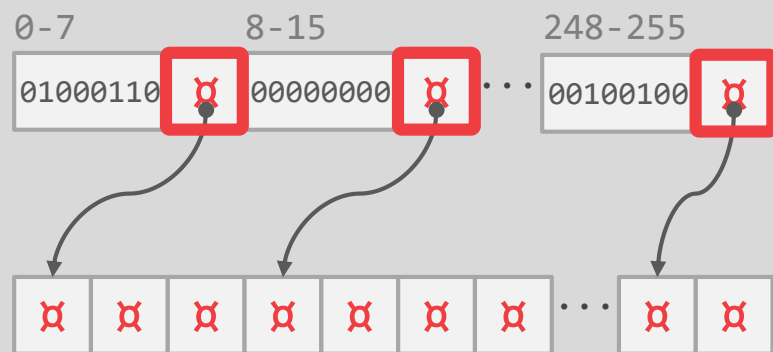
256-bit map to mark whether a prefix is present in node.

Bitmap is divided into eight segments, each with a pointer to a sub-array with pointers to child nodes.

JUDY ARRAYS: BITMAP NODES

Bitmap Node

Prefix Bitmaps *Sub-Array Pointers*



256-bit map to mark whether a prefix is present in node.

Bitmap is divided into eight segments, each with a pointer to a sub-array with pointers to child nodes.

JUDY ARRAYS: BITMAP NODES

Bitmap Node

Prefix Bitmaps Sub-Array Pointers



Child Pointers

256-bit map to mark whether a prefix is present in node.

Bitmap is divided into eight segments, each with a pointer to a sub-array with pointers to child nodes.

JUDY ARRAYS: BITMAP NODES

Bitmap Node

Prefix Bitmaps *Sub-Array Pointers*



Child Pointers

256-bit map to mark whether a prefix is present in node.

Bitmap is divided into eight segments, each with a pointer to a sub-array with pointers to child nodes.

ADAPATIVE RADIX TREE (ART)

Developed for TUM HyPer DBMS in 2013.

256-way radix tree that supports different node types based on its population.

→ Stores meta-data about each node in its header.

Concurrency support was added in 2015.



ART vs. JUDY

Difference #1: Node Types

- Judy has three node types with different organizations.
- ART has four nodes types that (mostly) vary in the maximum number of children.

Difference #2: Purpose

- Judy is a general-purpose associative array. It "owns" the keys and values.
- ART is a table index and does not need to cover the full keys. Values are pointers to tuples.

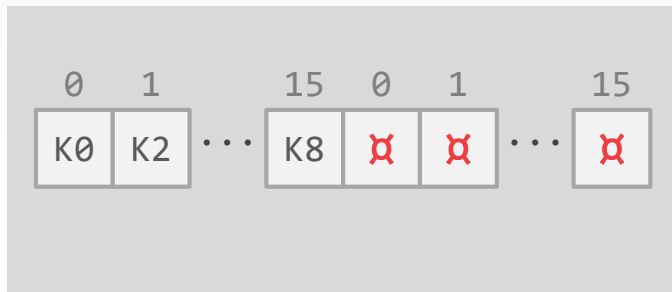
ART: INNER NODE TYPES (1)

Node4



Store only the 8-bit digits that exist at a given node in a sorted array.

Node16



The offset in sorted digit array corresponds to offset in value array.

ART: INNER NODE TYPES (1)

Node4



Store only the 8-bit digits that exist at a given node in a sorted array.

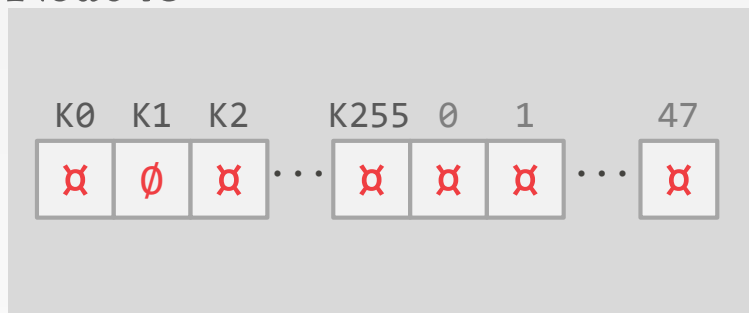
Node16



The offset in sorted digit array corresponds to offset in value array.

ART: INNER NODE TYPES (2)

Node48

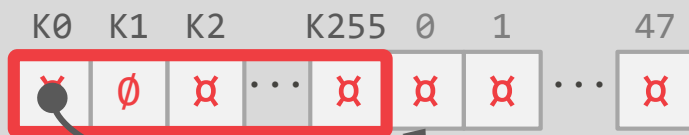


Instead of storing 1-byte digits, maintain an array of 1-byte offsets to a child pointer array that is indexed on the digit bits.

ART: INNER NODE TYPES (2)

Node48

Pointer Array Offsets

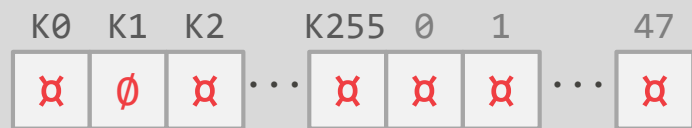


Instead of storing 1-byte digits, maintain an array of 1-byte offsets to a child pointer array that is indexed on the digit bits.

ART: INNER NODE TYPES (2)

Node48

Pointer Array Offsets



$256 \times 1\text{-byte} =$
256 bytes

$48 \times 8\text{-bytes} =$
384 bytes

640 bytes

Instead of storing 1-byte digits, maintain an array of 1-byte offsets to a child pointer array that is indexed on the digit bits.

ART: INNER NODE TYPES (3)

Node256



*$256 \times 8\text{-byte} =$
 2048 bytes*

Store an array of 256 pointers to child nodes. This covers all possible values in 8-bit digits.

Same as Judy Array's
Uncompressed Node.

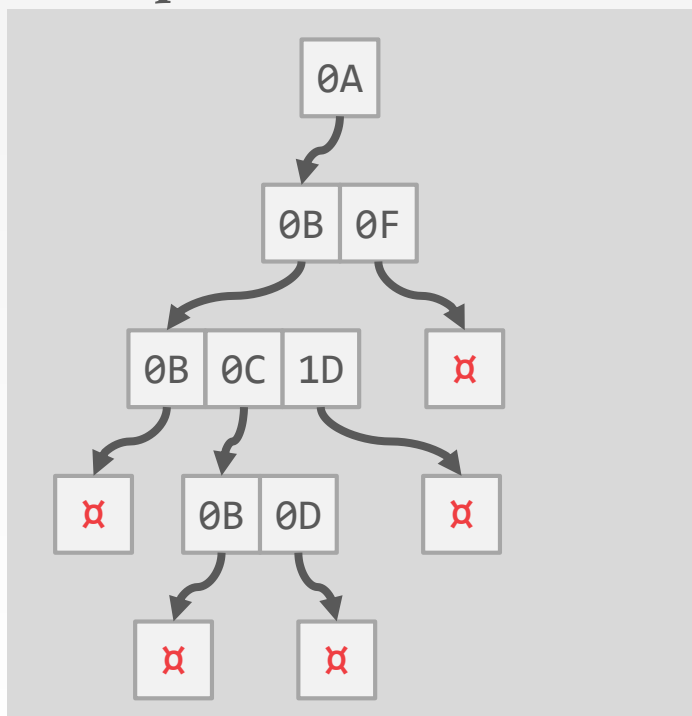
ART: BINARY COMPARABLE KEYS

Not all attribute types can be decomposed into binary comparable digits for a radix tree.

- **Unsigned Integers:** Byte order must be flipped for little endian machines.
- **Signed Integers:** Flip two's-complement so that negative numbers are smaller than positive.
- **Floats:** Classify into group (neg vs. pos, normalized vs. denormalized), then store as unsigned integer.
- **Compound:** Transform each attribute separately.

ART: BINARY COMPARABLE KEYS

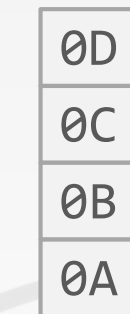
8-bit Span Radix Tree



Int Key: 168496141



Hex Key: 0A 0B 0C 0D



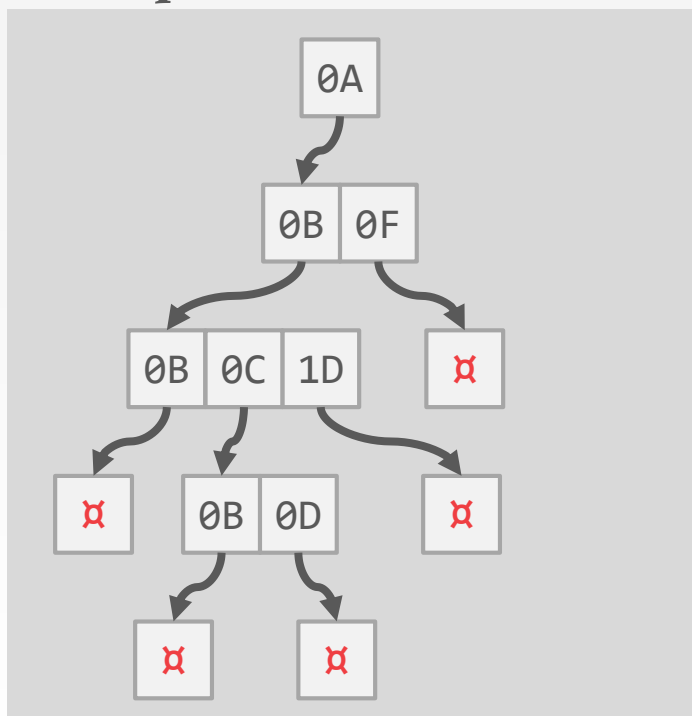
*Little
Endian*



*Big
Endian*

ART: BINARY COMPARABLE KEYS

8-bit Span Radix Tree



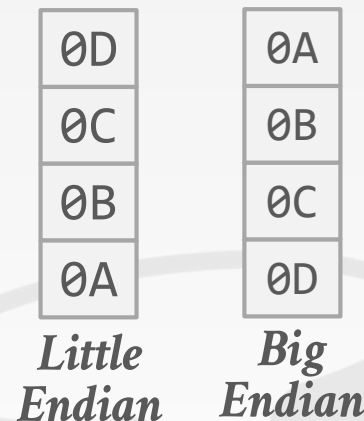
Int Key: 168496141



Hex Key: 0A 0B 0C 0D

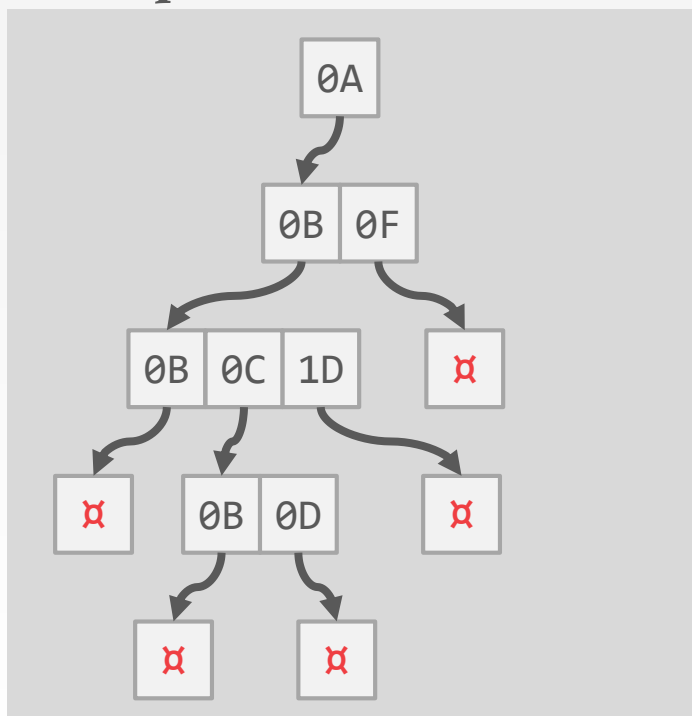
Find: 658205

Hex: 0A 0B 1D



ART: BINARY COMPARABLE KEYS

8-bit Span Radix Tree



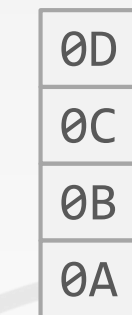
Int Key: 168496141



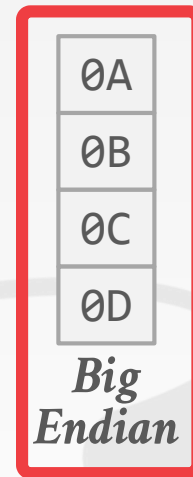
Hex Key: 0A 0B 0C 0D

Find: 658205

Hex: 0A 0B 1D



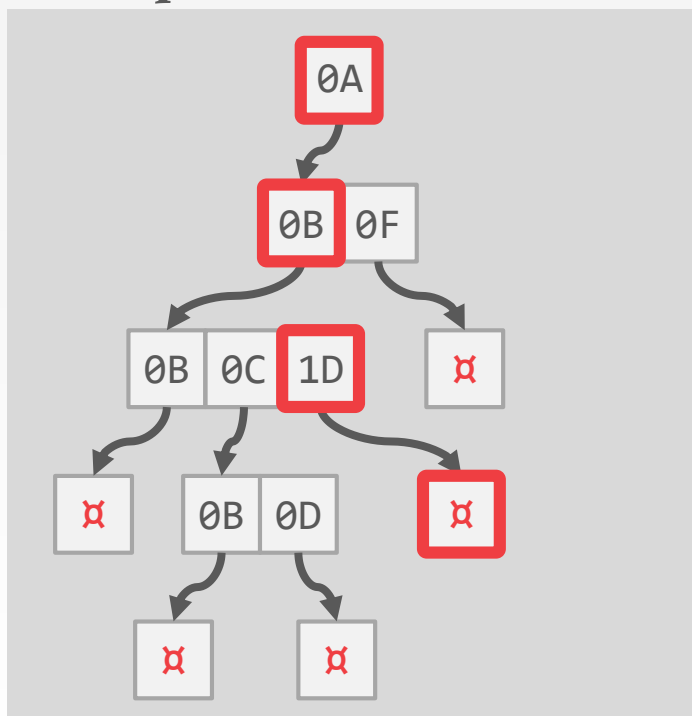
*Little
Endian*



*Big
Endian*

ART: BINARY COMPARABLE KEYS

8-bit Span Radix Tree



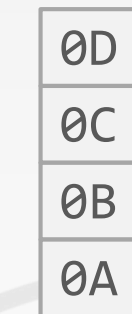
Int Key: 168496141



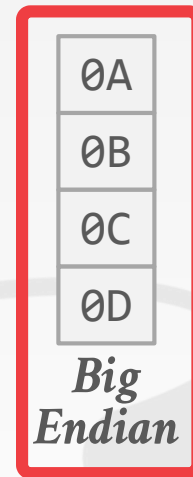
Hex Key: 0A 0B 0C 0D

Find: 658205

Hex: 0A 0B 1D



*Little
Endian*



*Big
Endian*

CONCURRENT ART INDEX

HyPer's ART is **not** latch-free.

→ The authors argue that it would be a significant amount of work to make it latch-free.

Approach #1: Optimistic Lock Coupling

Approach #2: Read-Optimized Write Exclusion



OPTIMISTIC LATCH COUPLING

Optimistic crabbing scheme where writers are not blocked on readers.

Every node now has a version number (counter).

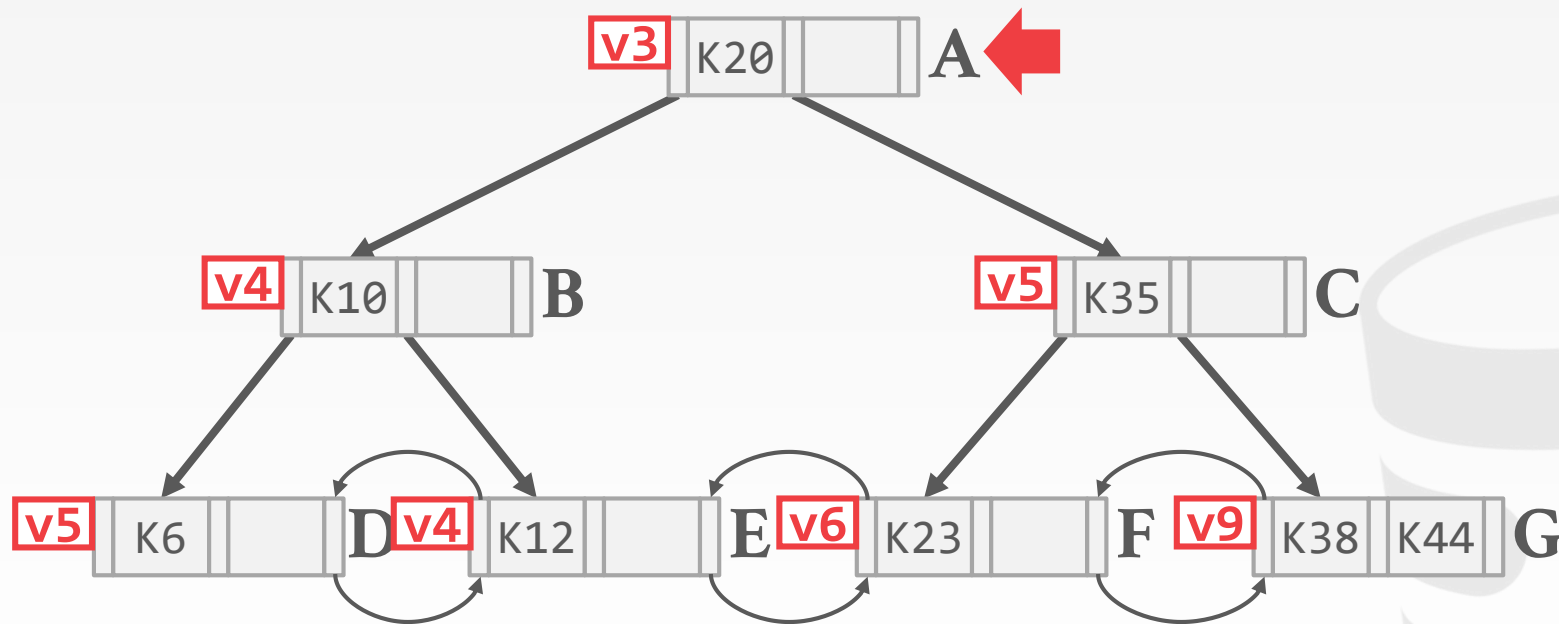
- Writers increment counter when they acquire latch.
- Readers proceed if a node's latch is available but then do not acquire it.
- It then checks whether the latch's counter has changed from when it checked the latch.

Relies on epoch GC to ensure pointers are valid.

OPTIMISTIC LATCH COUPLING

Search K44

A: Read v3
A: Examine Node

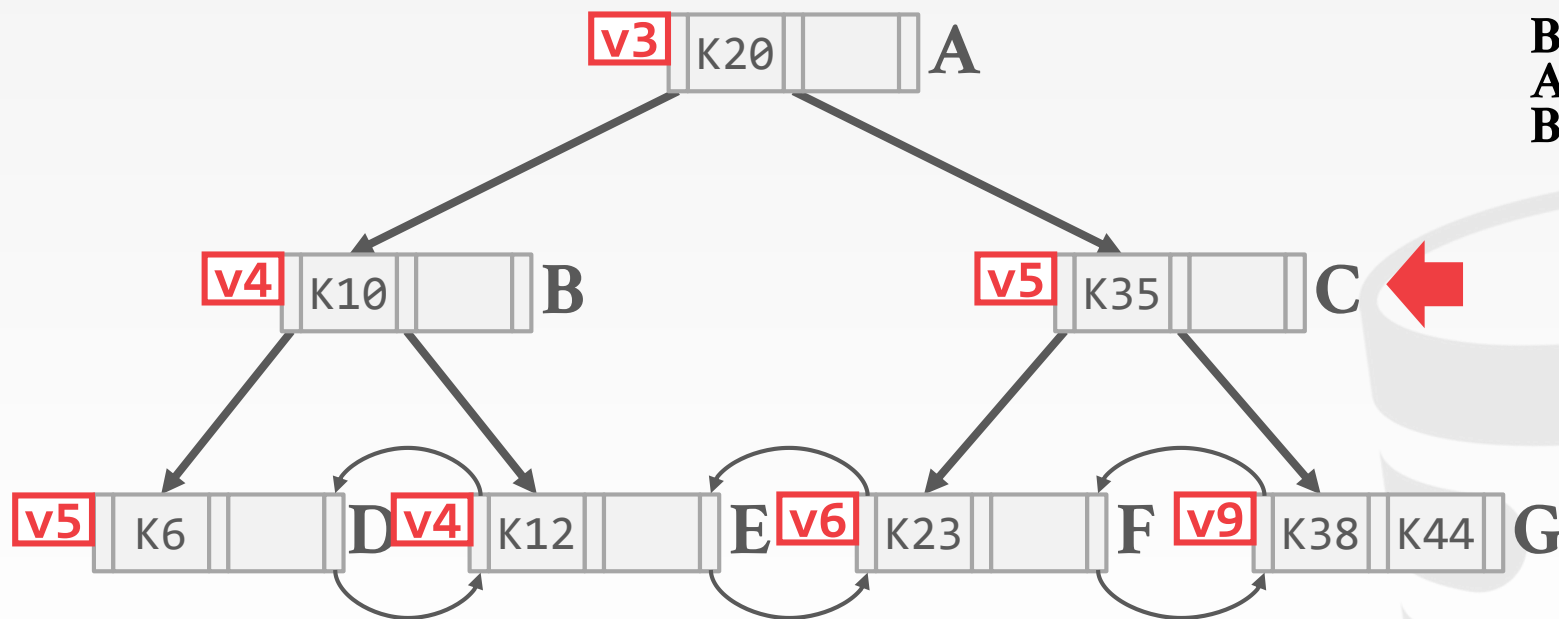


OPTIMISTIC LATCH COUPLING

Search K44

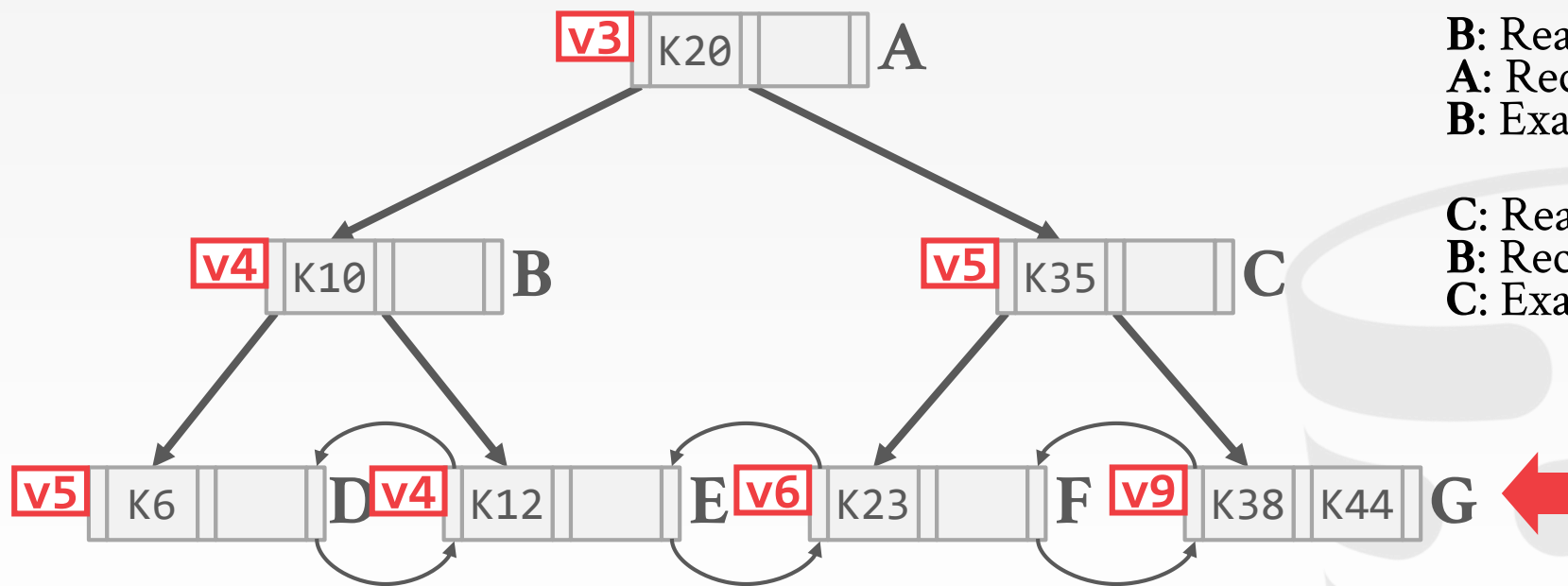
A: Read v3
A: Examine Node

B: Read v5
A: Recheck v3
B: Examine Node



OPTIMISTIC LATCH COUPLING

Search K44



A: Read v3
A: Examine Node

B: Read v5
A: Recheck v3
B: Examine Node

C: Read v9
B: Recheck v5
C: Examine Node

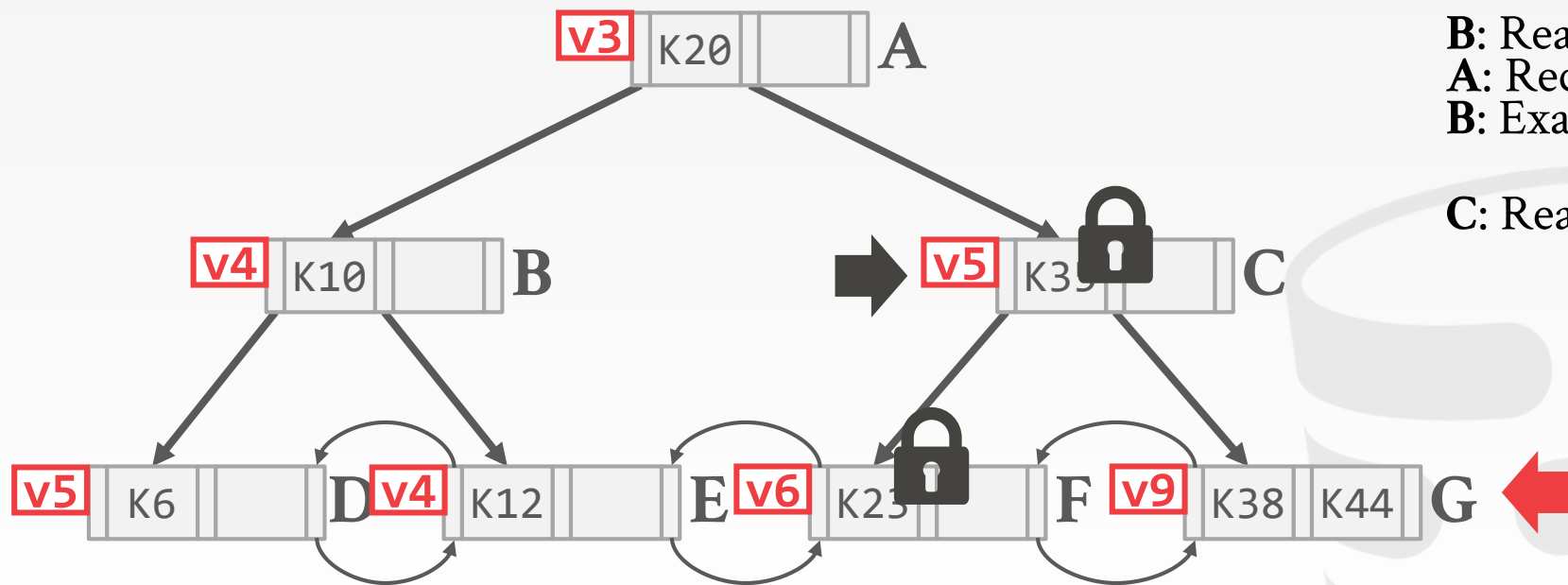
OPTIMISTIC LATCH COUPLING

Search K44

A: Read v3
A: Examine Node

B: Read v5
A: Recheck v3
B: Examine Node

C: Read v9



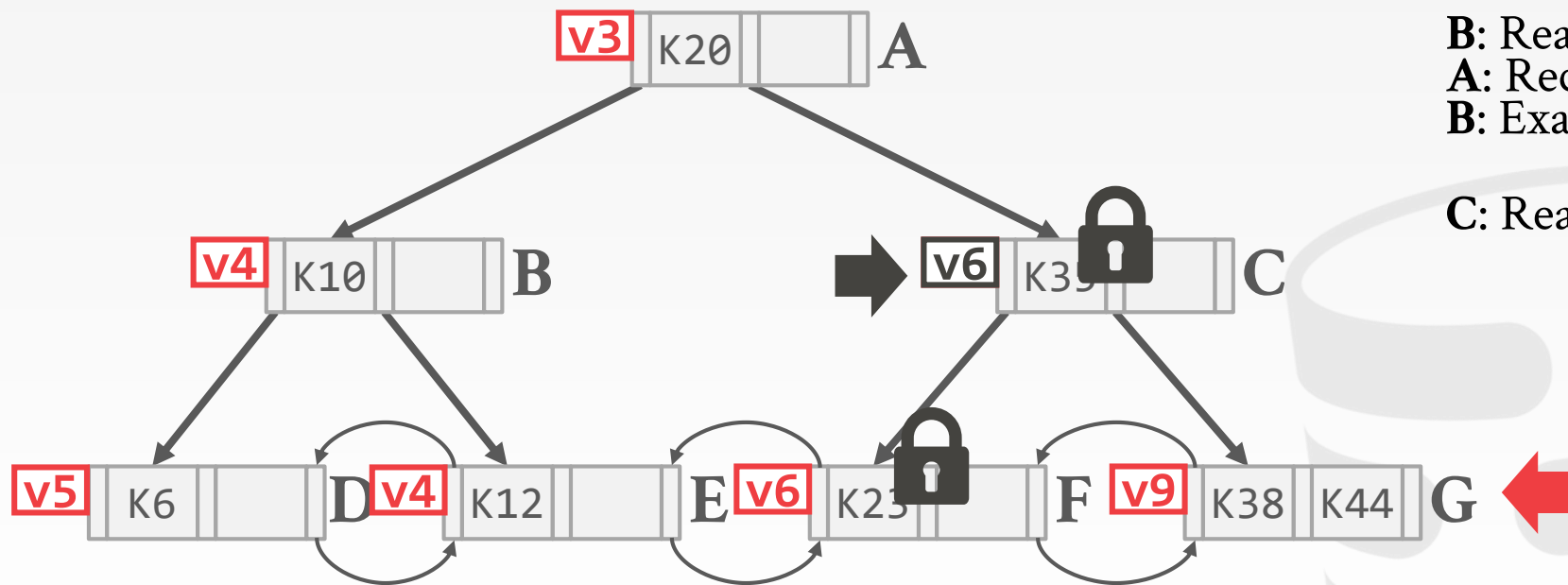
OPTIMISTIC LATCH COUPLING

Search K44

A: Read v3
A: Examine Node

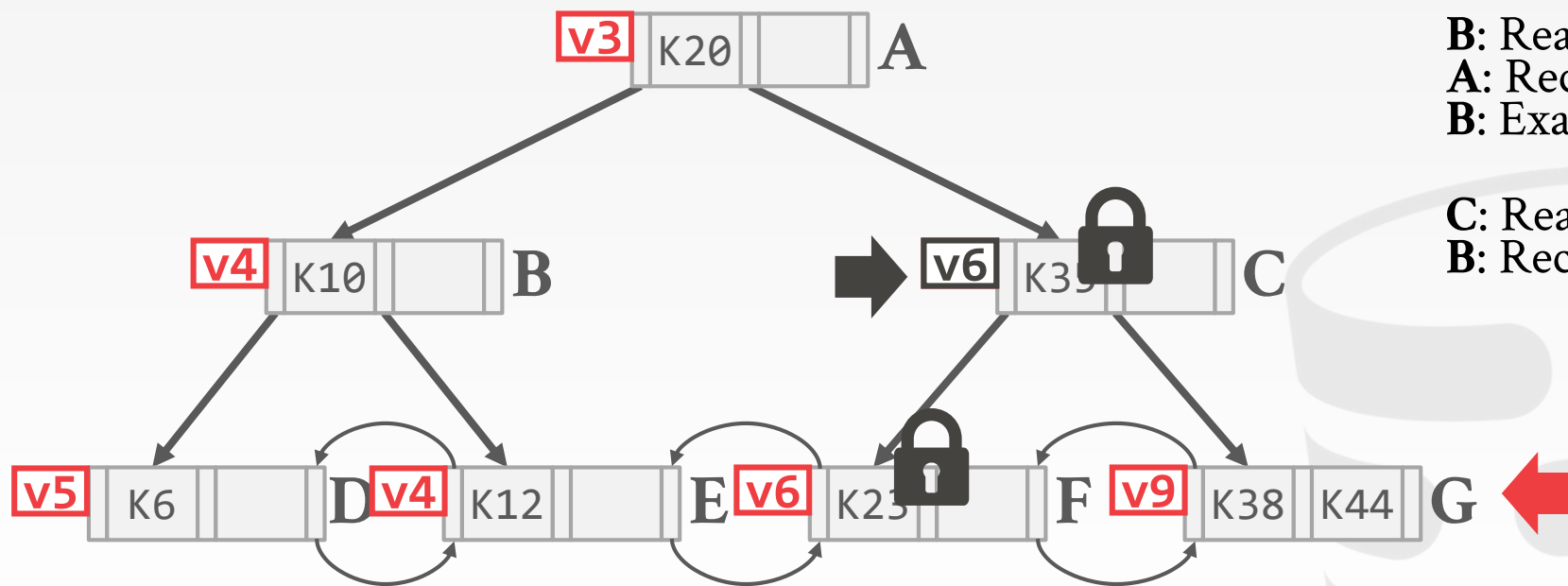
B: Read v5
A: Recheck v3
B: Examine Node

C: Read v9



OPTIMISTIC LATCH COUPLING

Search K44



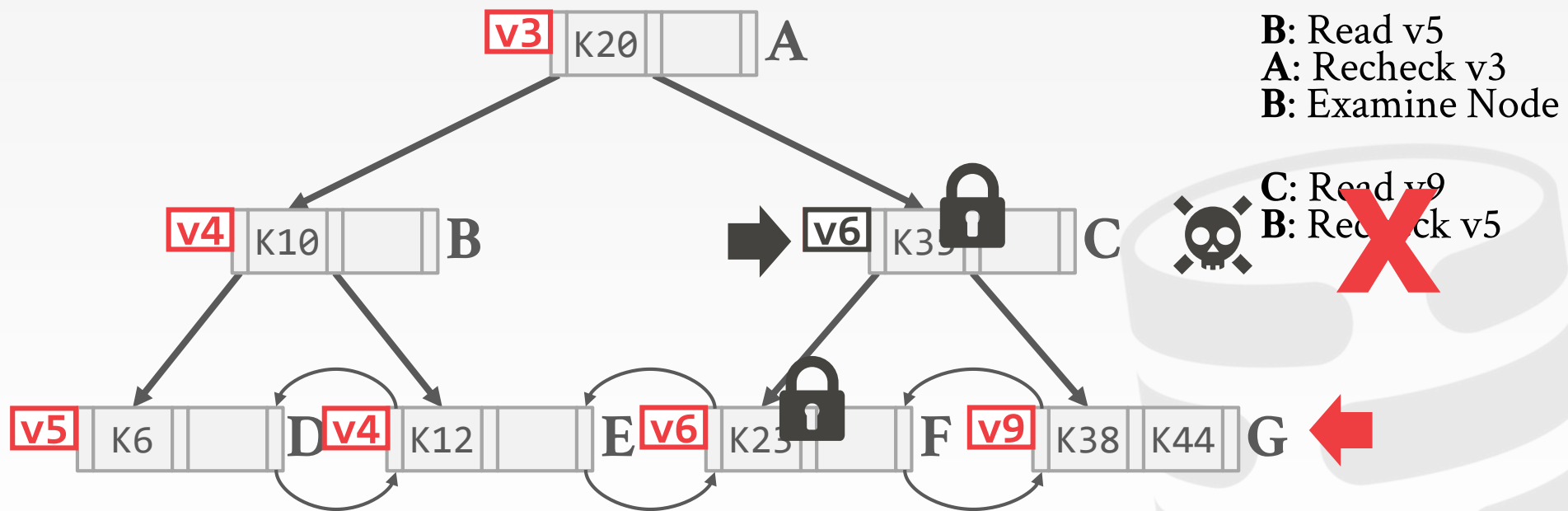
A: Read v3
A: Examine Node

B: Read v5
A: Recheck v3
B: Examine Node

C: Read v9
B: Recheck v5

OPTIMISTIC LATCH COUPLING

Search K44



READ - OPTIMIZED WRITE EXCLUSION

Each node includes an exclusive latch that blocks only other writers and not readers.

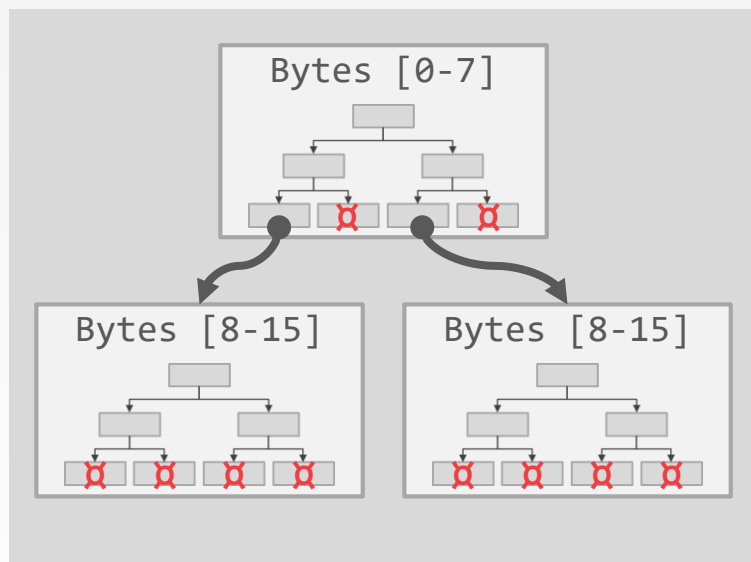
- Readers proceed without checking versions or latches.
- Every writer must ensure that reads are always consistent.

Requires fundamental changes to how threads make modifications to the data structure.

- Creating new nodes means that we have to atomically update pointers from other nodes (see Bw-Tree).

MASSTREE

Masstree



Instead of using different layouts for each trie node based on its size, use an entire B+Tree.

- Each B+tree represents 8-byte span.
- Optimized for long keys.
- Uses something similar to OLC.

Part of the Harvard Silo project.

IN-MEMORY INDEXES

Processor: 1 socket, 10 cores w/ 2×HT
Workload: 50m Random Integer Keys (64-bit)



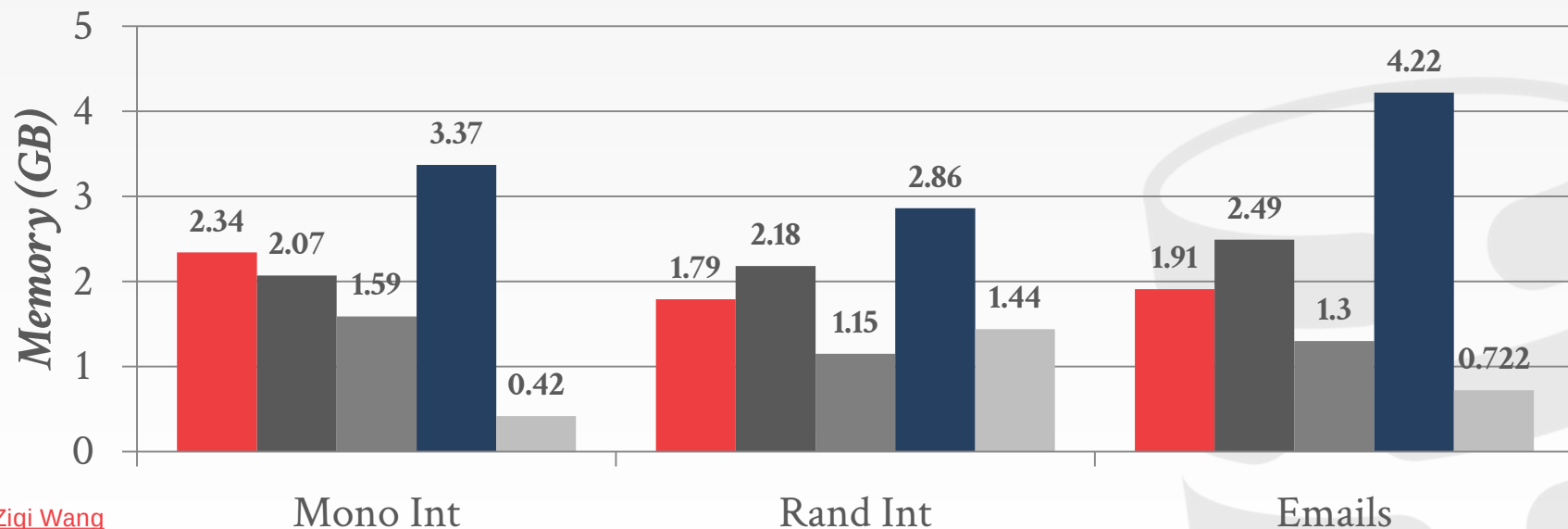
Source: [Ziqi Wang](#)

IN-MEMORY INDEXES

Processor: 1 socket, 10 cores w/ 2×HT

Workload: 50m Keys

■ Open Bw-Tree ■ Skip List ■ B+Tree ■ Masstree ■ ART



Source: [Ziqi Wang](#)

PARTING THOUGHTS

Andy was wrong about the Bw-Tree and latch-free indexes.

Radix trees have interesting properties, but a well-written B+tree is still a solid design choice.

NEXT CLASS

System Catalogs

Data Layout

Storage Models

